Data Staging on Untrusted Surrogates

Jason Flinn, Shafeeq Sinnamohideen, and M. Satyanarayanan

IRP-TR-02-02
May 2002

Intel Research Pittsburgh

Data Staging on Untrusted Surrogates

Jason Flinn[‡], Shafeeq Sinnamohideen[†‡], and M. Satyanarayanan[†‡]
[‡]Intel Research Pittsburgh and [†]Carnegie Mellon University

Abstract

In this paper, we show that untrusted computers can be used to securely facilitate mobile data access. We discuss a novel architecture, data staging*, that improves the performance of distributed file systems running on small, storage-limited pervasive computing devices by opportunistically caching data on nearby surrogate machines. Surrogates are untrusted and unmanaged: we use end-to-end encryption and secure hashes to provide privacy and authenticity of data, and have designed our system so that surrogates are as reliable and easy to manage as possible. Our results show that data staging improves the performance of interactive applications running on the Compaq iPAQ handheld by up to 64% .*

1 Introduction

Can an untrusted computer facilitate secure mobile data access? Surprisingly, the answer is "yes." In this paper, we show how such a computer can be used as a data staging node to improve the performance of cache miss handling in an Internet-wide distributed file system. The untrusted computer, called a *surrogate,* plays the role of a second-level file cache for a mobile client. By proactively staging data on the surrogate, cache misses from a nearby mobile client can be serviced at low latency (typically one wireless hop) instead of full Internet latency.

The use of surrogates for data staging can bridge a growing mismatch between the desires and expectations of users. On the one hand, users want the lightest and smallest wearable or handheld computer—for example, a wristwatch running Linux is no longer a fantasy [14]. On the other hand, users expect productivity improvements from mobile computing; ubiquitous access to personal and project data is a key part of this expectation. A distributed file system can provide such ubiquitous access, but requires crisp handling of cache misses to achieve good interactive user experience. For a small file, network latency to a distant file server on the Internet is typically the dominant component of cache miss service time. This can be reduced by redirecting cache misses to staged data on a nearby surrogate while still maintaining the consistency guarantees of the underlying file system. The alternative of totally avoiding cache misses through hoarding [8, 11] is not viable because of limited cache space and the need to view recent updates by other users.

What is the likelihood of a mobile computer finding a nearby surrogate? Although the chances are low today, we predict that continuing decline in mass-market hardware prices will improve these chances in the future. Desktop computers at discount stores already sell today for a few hundred dollars, with prices continuing to drop. In the foreseeable future, we envision public spaces such as airport lounges and coffee shops being equipped with surrogates for the benefit of customers, much as comfortable chairs and table lamps are provided today. These will be connected to the wired Internet through high-bandwidth networks, and to mobile clients in their neighborhood through wireless technologies such as 802.11 or Bluetooth.

Since hardware cost is only a small part of the total cost of ownership of a system, it is essential that surrogates require virtually no maintenance or system administration. Like a chair or table lamp, they should require negligible attention after initial setup. Only then will they be cheap enough for widespread deployment. This leads to two important assumptions about surrogates in our work: they are *unmanaged* and *untrusted.* In particular, we make surrogates as reliable and easy to manage

as possible by maintaining no hard-state on surrogates, building as much as possible on commodity software, and pushing functionality from surrogates to client and server machines.

We rely on the concept of *caching trust* to guard against malicious surrogates. This end-to-end approach ensures privacy through encryption, and integrity through verification of secure hashes. Even the most resource-challenged mobile client typically has enough disk or flash storage to cache hashes and encryption keys of all files of potential interest to the user. Hence, data never has to be stored in the clear on a surrogate — it is encrypted at the server and remains encrypted on the surrogate. When servicing a cache miss, the client decrypts file data from the surrogate, calculates its hash, and verifies the computed hash against its cached copy. A compromised surrogate could, of course, still cause mischief through denial of service. The client's only recourse is to directly contact servers. Even in this case, performance is no worse than in the absence of data staging.

In this paper we report on the feasibility of data staging on untrusted surrogates. Our prototype implementation is based on the Coda file system [17], but is structured for easy use with other distributed file systems. Measurements from this prototype confirm the performance benefits of data staging. For bursty, short-term workloads, data staging reduces the cumulative delay due to file operations by up to 77%. We have confirmed these results by replaying long-term traces of file-system activity—these experiments show average reductions in file operation latency of 59%.

The focus of this paper is on the file system aspects of data staging. Topics such as surrogate discovery (possibly through mechanisms such as JINI [22]) and load balancing across multiple surrogates are outside its scope. We begin, in Section 2, with an example scenario to illustrate use of our system. In Section 3 we describe the design and implementation of data staging. The following section describes additional scenarios under which data staging can be profitably employed. We evaluate the benefit of data staging for storage-limited clients in Section 5. The final two sections discuss related work and summarize the key points of our work.

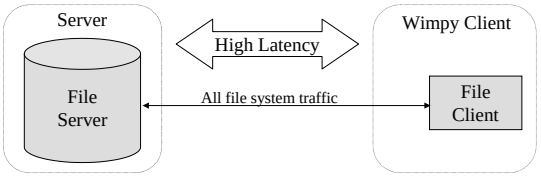
2 Usage Scenario

What kind of impact can proactive data staging have in real life? To illustrate our vision, we describe an imaginary scenario in this section. For ease of exposition, we assume that support facilities such as surrogate discovery and location sensing are provided by Aura [16, 6].

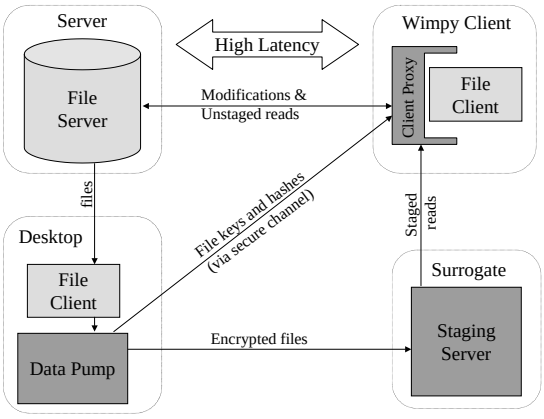
Joe, a Pittsburgh-based Intel engineer, is on a business trip to San Jose. His morning and early afternoon have been spent in back to back meetings for a new product. These meetings generated a number of urgent questions. To answer them, Joe will have to consult many documents containing design specifications and test results of an old product.

As Joe walks to his next meeting, he sees a Starbucks coffee shop and decides to take a break. The coffee shop has 802.11 wireless coverage and provides a number of surrogates for use by its customers. As Joe walks in, Aura discovers these resources and is assigned a surrogate. Since Joe's schedule indicates that his next meeting is not for another 20 minutes, Aura guesses that he will take a break. While Joe's Mocha Frappucino is being made, all 10 GB of his file data are staged on the surrogate. By the time Joe sits down and pulls out his handheld computer, all the data he will access is already staged nearby. He is able to effectively use an interactive data visualization tool to explore a number of alternatives. Had his data not been staged nearby, the delay loading data would have made the tool's interactive performance miserable and he would have just given up. Because of staging, he is blissfully unaware of the Internet delays. In just a few minutes he converges on a solution, and sends email to report his findings to his anxious colleagues.

Refreshed, Joe leaves for his next meeting. As he departs from the coffee shop, Aura deletes the staged data and ends use of the surrogate.



(a) Without data staging



(b) With data staging

Figure 1. Data staging architecture

3 Design and Implementation

3.1 Overview

Figure 1(a) shows a typical scenario that motivates the need for data staging. An interactive application running on a storage-limited client accesses files stored in a distributed file system. The file system attempts to reduce access time by caching files on the client machine, but space limitations prevent it from caching all but a portion of the files that the user might potentially read. Consequently, many files needed by the application are not cached, and must be fetched from the distant file server. Because the application reads multiple files sequentially and reading each file incurs multiple round-trip delays, the user experiences many frustrating delays.

Figure 1(b) shows how data staging improves this scenario. On the client machine, we interpose a proxy that intercepts file system traffic. When the proxy observes that remote file accesses are incurring high latency, it finds a surrogate in the nearby network environment that is willing to provide extra storage capacity. The proxy registers with the surrogate and stages the set of files that the user is most likely to access in the future; because the surrogate has much greater storage capacity than the mobile device, the set of files that can be stored on the surrogate is much greater than the set that can be stored on the client.

Staging is expedited by a *data pump*, typically located near the file server. When the client proxy wishes to stage a file, it sends a message to the pump through a secure channel. The pump authenticates the message, reads the file from the file system, encrypts the file, and generates a cryptographic hash of the data. The pump transmits the encrypted file to the surrogate and sends the file key and hash to the client through the secure channel. When a staged file is read by the application, the proxy fetches the file from the nearby surrogate, decrypts it, and uses the hash to verify that the file has not been modified. Use of the surrogate decreases the number of high-latency file accesses and dramatically reduces the number of long delays experienced by the user.

The proxy-based architecture allows us to achieve a great deal of independence from the underlying distributed file system. Data staging requires no modification to file system source code; we use gray-box techniques where necessary to infer file system state. Further, almost all file-system specific code is encapsulated within the client proxy. Thus, while our current system uses Coda as the underlying file system, the changes needed to support additional file systems such as NFS and AFS would be minimal.

In the next three subsections, we describe the design and implementation of the surrogate, client proxy, and data pump in more detail.

3.2 Surrogate

We are convinced that widespread deployment of surrogates hinges on ease of management. As mentioned previously, surrogates should be as easy to manage as table lamps—they should not need a system administrator or a complex user manual.

```
SurrogateRegister    (IN surrogate, IN pubkey, OUT clientid, OUT quota,
                     OUT sesskey);
SurrogateDeregister  (IN surrogate, IN clientid);
SurrogateStage       (IN surrogate, IN clientid, IN fileid, IN buf, IN buflen);
SurrogateUnstage     (IN surrogate, IN clientid, IN fileid);
SurrogateGet         (IN surrogate, IN clientid, IN fileid, IN buf, IN buflen,
                     IN key, IN hash);
```

Figure 2. Surrogate API

We have identified two design principles that improve ease of management. The first principle is to build as much as possible upon widespread commodity software, so as to leverage the improved reliability that comes through the extensive testing provided by a large user community. To this end, we use the Apache Web server as the base system for our surrogates. We have identified the minimum set of additional functionality that must be located on the surrogate, and provide this functionality with CGI scripts. All other functionality is pushed to the client proxy and data pump in order to keep the custom code base on the surrogate as simple and reliable as possible.

The second design principle is to maintain no long-term state on the surrogate. For example, we do not buffer client modifications to file data on surrogates. Since all state is soft, no critical information is lost if the surrogate is disrupted by power failure or a system crash. The only consequence of surrogate failure is that clients receive the same sub-par performance for file accesses that they would have received if the surrogate had not been present in the first place.

Figure 2 shows the surrogate API. C wrapper libraries on the client and data pump implement these functions as HTTP/1.1 operations. When `SurrogateGet` is called, the wrapper library issues a HTTP GET request; the remaining functions are implemented as POST operations that invoke CGI scripts on the surrogate.

A client calls `SurrogateRegister` to start using a surrogate. The client provides its public key, which is used for authentication. If the surrogate is willing to provide storage space to the client, it assigns the client a unique identifier and specifies a storage quota. The surrogate also generates a session key, encrypts it with the client's public key, and returns it to the client. The session key is used to authenticate all subsequent messages that modify what data is staged on the surrogate. `SurrogateStage`, `SurrogateUnstage`, and `SurrogateDeregister` each send the surrogate a token encrypted with the session key that represents the command being executed; nonces are used to guard against replay attacks.

The `SurrogateStage` function places files on the surrogate. The surrogate treats each file as a binary chunk of data with an identifier unique to the client. The CGI script for `SurrogateStage` stores the file data and updates the amount of storage currently used by the client. However, if storing the file would cause the client to exceed its quota, an error is returned. If a file with the same identifier already exists on the surrogate, the previous data is replaced and the quota updated appropriately.

Once a file is staged, it may be retrieved by the client using the `SurrogateGet` function. Alternatively, the client may delete the staged file to free up storage capacity with the `SurrogateUnstage` function. The final function, `SurrogateDeregister`, is used to explicitly release surrogate resources. Whenever a client deregisters, the surrogate deletes the files that it has stored and allows its previously allocated storage capacity to be used by new clients.

3.3 Client proxy

The client proxy performs multiple tasks in the data staging architecture. Its primary task is intercepting file system traffic and redirecting appropriate requests to surrogates.

However, the client proxy is also responsible for determining which files to stage on a surrogate. It

periodically recomputes the set of files most likely to be accessed and arranges for as many as possible to be staged on the nearby surrogate. The client proxy's final task is maintaining a consistent view of file data. Whenever the client modifies data that has been staged on the surrogate, or it receives a *callback* notification from the file server that another client has modified the data, it marks the staged file as invalid. This allows data staging clients to maintain the same consistency guarantees provided by the underlying file system.

3.3.1 Intercepting file system requests

The client proxy intercepts all traffic bound for a specified set of servers. It masquerades as a local file server; thus, the file system client believes it is connected to a file server running on the local machine. When the proxy receives a request from the file system client, it either transparently forwards it to the distant file server for which it is masquerading, or it retrieves staged data from a nearby surrogate and responds to the request itself. In the presence of multiple file servers, this design allows us to interpose proxies only for high-latency servers—latency to nearby servers need not be adversely affected.

The proxy maintains a hash table of all files stored on the surrogate. When it intercepts a request to read data, it checks whether a valid copy of the file data is currently staged on the surrogate. If the file is staged and valid, it calls `SurrogateGet` to retrieve the data, decrypts the file, computes a secure hash, and verifies that the hash matches its cached value. The file retrieval, decryption, and hash computation are pipelined to reduce access latency. If a valid copy of the file is not staged, the proxy forwards the request to the distant file server.

All modifications to file data by the client are sent directly to the file server. If a copy of the modified file is staged, the proxy marks the copy as invalid. A new copy of the modified file may later be staged as described in Section 3.3.3.

One alternative design would be to send modifications to the nearby surrogate, and have the surrogate reintegrate the changes with the file server. However, this conflicts with our desire to use untrusted surrogates that are as simple and maintainable as possible. Clients would need to guard against malicious or faulty surrogates which might lose modifications. Additionally, surrogates would need to run potentially complex reconciliation protocols.

3.3.2 Predicting file accesses

The ability to successfully predict future accesses is essential to any caching strategy. In data staging, the client proxy observes file system traffic, logs the files that are accessed, and uses the data to predict what files will be fetched in the future. The proxy takes advantage of Coda's Mariner interface, which reports the name of each file upon invocation of the `open` system call. With an underlying file system that does not provide this level of detail, the proxy would only be able to observe accesses which miss in the cache, providing slightly less accurate information.

The proxy uses a simple LRU prediction strategy. It maintains a list of files ordered by last access time. When it discovers and registers with a new surrogate, it stages files at the head of the LRU list on the surrogate until the quota is reached. Prediction and staging are performed by dedicated threads, allowing these activities to proceed in parallel with the proxying of file system traffic. This LRU approach is quite similar to the caching strategy of the underlying Coda file system. However, the file access data gathered by the proxy could potentially by used to implement alternative prediction strategies [9, 11].

Our prediction implementation is based upon two important observations. First, we expect available cache space to change by several orders of magnitude as users move: from a few tens of megabytes when no surrogate is available, to gigabytes of storage when a surrogate is located nearby. Therefore, the amount of prediction information maintained should be independent of the current cache size. We maintain LRU information sufficient to populate the allocated quota of any surrogate we may encounter in the future.

Our second observation is that it is now common for one person to access the same data on multiple machines; for example, a single user may own a home

computer, a desktop at work, a laptop, and a handheld device. If the user has recently accessed a file on one machine, it is more likely that the user will soon access the file on other machines. We account for this behavior by storing per-client LRU information in the distributed file system itself. When the proxy starts, it combines the LRU data from all machines that the user has recently accessed to generate a global LRU ordering. This ordering is then used to select which files are staged on a surrogate. To minimize update traffic, the client proxy reads and writes LRU information on a periodic basis (currently every 60 minutes). While this strategy has the potential to lose some data in the event of system crash, the effect of such a loss would only be a slight decrease in prediction accuracy.

A secondary benefit of storing LRU information in the distributed file system is that the LRU data does not consume precious client resources during normal operation. While the storage requirements for the LRU data are not large (< 1 MB in our experience), this may still represent a significant portion of the storage capacity of a small client.

3.3.3 Maintaining consistency

The client proxy is the final arbiter of whether data staged on a surrogate is valid. It associates a valid bit with each file in its hash table of staged files. After successfully staging a file, this bit is set to valid. When the proxy intercepts a request that modifies file data, it searches for the file in the hash table and invalidates the entry if found. When another client modifies a staged file, Coda provides a `callback` notification to all clients which have accessed the file. The data pump receives this callback and forwards it to the client proxy. If the modified file is currently staged, the proxy invalidates it.

The proxy periodically rescans the LRU list to warm the surrogate cache. It stages any file at the head of the LRU list which is marked invalid or is not currently staged. Files further towards the tail of the LRU list are unstaged to make room within the allocated quota. We have chosen to make the surrogate cache inclusive with the client's file cache, approximating the stack property. This means that files which are evicted from the client cache will immediately be available in the surrogate cache. The performance penalty of inclusive caching is small, since the surrogate will typically have a cache size several orders of magnitude greater than the client.

An alternate approach would be to immediately stage files after they are invalidated or newly created. However, when file modifications are bursty, this alternate approach would lead to many successive stages and invalidations, wasting client bandwidth and energy.

3.4 Data pump

The data pump fetches and stages files on behalf of the client proxy. Typically, the data pump will be located in close network proximity to the file server. Potentially, it may run on the file server itself. More often, however, we expect that each user will run a data pump on their desktop computer. The latter alternative has the benefit of reducing load on file servers; since the desktop will have a large file cache, most requests to stage data can be serviced without contacting the file server.

Client proxies contact the data pump and establish a secure tunnel for communication. The two parties use public key cryptography to establish a symmetric session key. We use the symmetric session key to encrypt all further traffic because symmetric key encryption is less computationally demanding than public key encryption. Public key distribution is simplified since a single user will typically operate both machines (if the data pump is located on a desktop), or both machines will lie within the same administrative domain (if the data pump is located on a file server).

When the client proxy needs a file to be staged, it sends the data pump the fully-qualified path name and the IP address of the surrogate. The data pump retrieves the file from the underlying distributed file system, generates a random key, encrypts the file, and generates a cryptographic hash of the file data. The pump calls `SurrogateStage` to place the encrypted file on the surrogate. If successful, the pump sends the encryption key and hash to the surrogate where they are stored for later reference. By default, our current implementation uses 64 bit DES encryption and generates 64 byte MD5 digests

of file data. The client storage requirement of 72 bytes per file is thus significantly less then the average file size reported in file system studies [4, 21].

The client proxy is multi-threaded, allowing it to effectively overlap computation and network transmission, and to concurrently service the requests of multiple clients. As a performance enhancement, the proxy may batch multiple `SurrogateStage` requests into a single HTTP/1.1 POST request; this decreases the time needed to stage a large number of small files.

4 Further benefits of data staging

To date, our work has focused on exploring the benefits of data staging for storage-limited clients in pervasive computing environments. Yet, we believe that data staging will prove desirable in several other important scenarios.

The use of Infostations [23] has been suggested as a solution for overcoming the bandwidth limitations of wide-area wireless networks. Clients periodically pass through short-range, high-bandwidth zones located within the pervasive infrastructure. Data updates such as file modifications can be burst transmitted to the client during the short period of high-bandwidth connectivity.

We believe that the data staging architecture will prove to be an ideal mechanism for caching file data in Infostation-like environments. For example, consider a researcher who is traveling across the United States by plane with a stopover in Chicago. During the trip, the researcher works with files associated with an ongoing project; these files are concurrently being updated by other researchers around the country. As she hurries between gates in Chicago to catch her next flight, the researcher's mobile computer contacts a nearby surrogate and burst downloads modifications so that an updated copy is available during the next flight. Because the data is staged on a surrogate connected to a high-bandwidth network, the entire set of data can be downloaded in the short time available.

Data staging also has large potential benefits for battery-powered clients. Studies of energy usage show that an 802.11 network interface represents a very large portion of the total energy budget of small handheld devices [5, 19]. Under periods of high network usage, such devices quickly run out of battery power.

Our data staging architecture can significantly extend the battery lifetime of mobile clients in two ways. First, fetching small files from surrogates is significantly faster than fetching the equivalent data from distant file servers. Since the network is active for shorter periods with data staging, the network interface can be put into longer and deeper power saving modes.

A second, potentially greater, benefit is that the speculative prefetching of data to clients can be dramatically reduced. With data staging, the latency of a cache miss is much lower. Therefore, the client file system can afford to be less aggressive in keeping the cache up-to-date with the files the user is most likely to fetch. Instead, the client can stage such files on a nearby surrogate, knowing that the penalty of a cache miss will be small.

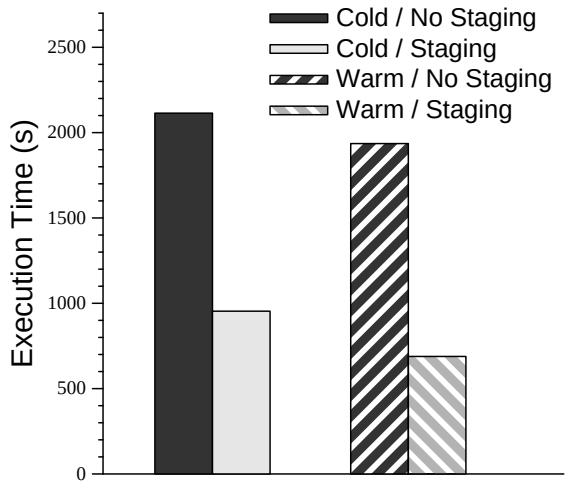
5 Evaluation

How much does data staging improve the performance of interactive applications running on storage-limited clients?

We answer this question by measuring the performance impact of data staging in two different usage scenarios. In the first scenario, we mirror the short-term bursty activity described in Section 2. We model the activities of a user browsing images from a large image library stored in the Coda file system and examine the potential performance benefits of data staging. In the second scenario, we examine the benefits of data-staging over a longer time period. We use recorded traces of client file system workloads to represent the activities of a user on a multi-day visit to a distant work location and examine how data staging reduces file operation latency.

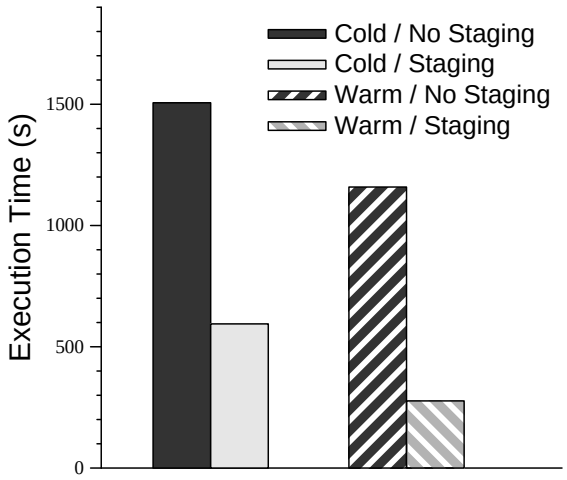
5.1 Experimental setup

We ran a Coda server on a powerful desktop computer with 2 GHz Pentium 4 processor and 512 MB DRAM. We also chose to run the data pump on the same machine since scalability was not a focus of



This figure shows the benefits of data staging for the image benchmark running on a Compaq iPAQ handheld with a 16 MB file cache. Each set of bars shows the total execution time for the benchmark. The leftmost two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches.

Figure 3. Image trace on iPAQ handheld



This figure shows the benefits of data staging for the image benchmark running on a client with a 32 MB file cache. Each set of bars shows the total execution time for the benchmark. The leftmost two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches.

Figure 4. Image trace with 32 MB cache

our evaluation. The surrogate ran on an identical desktop computer—we used NISTnet [2] to emulate a 30 ms. delay (60 ms. round-trip time) between the Coda server and surrogate. The 30 ms. delay is typical of current coast-to-coast delays in the United States. The client machine in our experiments used a 11 Mb/s 802.11 wireless network card for communication. The wireless hub was on the same network segment as the surrogate; we also emulated a 30 ms. delay between the client and Coda server.

For these experiments, we used two client machines. The first was a Compaq iPAQ handheld with 64 MB DRAM, 32 MB flash, and 206 MHz StrongArm processor. Resource limitations restricted us to a 16 MB client cache on this platform. The second client was an IBM ThinkPad T20 laptop with 700 MHz Pentium III processor and 128 MB DRAM. Although not typical of the storage-limited clients for which data staging is targeted, this platform allowed us to explore several different cache sizes in a stable development environment.

5.2 Image browsing

5.2.1 Methodology

We first constructed an image browsing benchmark that emulates the bursty workload described in Section 2. Over a roughly one month period, we recorded accesses to a 477 MB library of digital photographs stored in the Coda file system. From the access log, we selected a segment comprised of the first 9000 file operations. The typical client activity captured in this segment first opens a large number of small, thumbnail images in a directory, then opens a smaller number of large, full-sized images in the same directory. The selected trace segment reads 87 MB of data, approximately 18% of the library.

We replayed the trace as fast as possible using the DFSTrace tool [12], which performs file operations identical to those recorded in the trace. The figure of merit is the time needed to execute the complete trace—this corresponds to the total delay that the user experiences while loading images. This is distinct from the total amount of time the user will take to view the images, which will include a variable amount of think time. Although think time does not affect the total delay without data staging, the benefits of staging would increase with think time since

Trace	Number of Operations	Length (Hours)	Write Operations	Working Set (MB)
purcell	87739	55.32	6%	254
messiaen	44027	42.54	2%	227
robin	37504	30.92	7%	85
berlioz	17917	15.70	8%	57

This figure shows the file system traces used for our evaluation. Since data staging currently uses Coda as the base file system and Coda uses the `open-close` semantics of AFS, individual `read` and `write` operations are not included. Hence, "write operations" refers to activities such as `close` after `write` and `mkdir`. The working set is the total size of the files accessed during a trace.

Figure 5. File traces used in evaluation

the client could stage data during those intervals.

The effectiveness of any caching strategy depends upon how well one can predict which data will be accessed in the future. In data staging, prediction accuracy impacts performance twice: first, in the hit rate of the local file system cache, and second, in the hit rate of staged data on the surrogate. We examine the effect of prediction accuracy in our evaluation by looking at the two extreme ends of the spectrum. In the `cold` scenario, no data is contained in the Coda client cache when we begin playing a trace. In the `warm` scenario, we fill the entire Coda client cache with the set of files initially accessed by the trace.

For both scenarios, we compare total file access time with and without data staging. In the `warm` scenario, we stage all files referenced by a trace on the surrogate before executing the trace. In the `cold` scenario, the client discovers the surrogate as the trace begins. Thus, staging of data on the surrogate proceeds concurrently with trace replay. Files are staged in random order—this is somewhat of a pessimistic assumption, presumably LRU ordering would cause those files most likely to be accessed to be staged first.

5.2.2 Results

Figure 3 shows the results of running the image benchmark on the iPAQ handheld. In the `cold` scenario, data staging reduces the total time to execute the trace by 55% (19:20 minutes). The `warm` scenario executes in less time because the images initially viewed by the user are already in the client cache. Under this assumption, data staging also yields excellent performance improvement, reducing execution time by 64% (20:47 minutes).

Figure 4 shows results for the T20 laptop with a 32 MB client file cache. Data staging is slightly more effective on this platform, reducing total benchmark execution by 61% (15:12 minutes) in the `cold` scenario and 77% (14:42 minutes) in the `warm` scenario.

5.3 Long-duration file traces

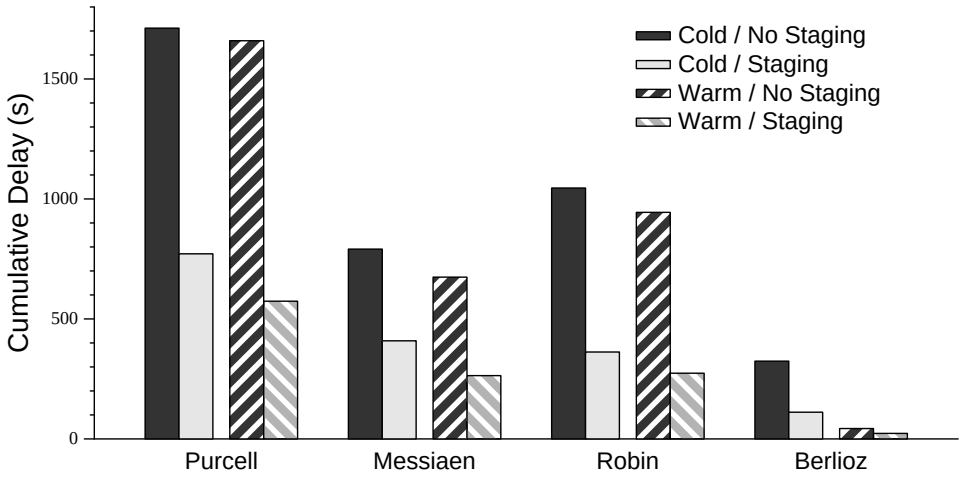
5.3.1 Methodology

To emulate the activity of a user on a multi-day visit to a distant work location, we replayed four different traces of client file system activity. We selected these traces, which are summarized in Figure 5, from the DFSTrace dataset gathered by Mummert et al. [12] at Carnegie Mellon University. Each trace was gathered on a different computer between 1991 and 1993; their duration ranges between 15 and 55 hours.

We used DFSTrace to replay each file trace. We repeated the methodology of section 5.2.1 by examining performance with and without data staging in both the `cold` and `warm` scenarios. The figure of merit is the total time needed to perform all file operations in the trace; this is equivalent to the amount of delay the user would experience during the trace period while waiting for file operations to complete.

We replay recorded inter-request delays for the first 15 minutes of each trace for the `cold` scenario—during this time, the client stages data on the surrogate while it concurrently proxies file operations. After 15 minutes, we replay the remainder of the trace as fast as possible, allowing us to complete the experiments in a reasonable amount of time. Elimination of delays does not affect the time to service file operations without data staging. With data staging the results are somewhat pessimistic, since the client does not restage data that has been invalidated due to modifications before the trace completes.

Although we assume here that all file activity is the result of foreground activity, it is likely that some of the trace activity was generated by background processes; however, this information is very difficult to distinguish from the trace data.



This figure shows the benefits of data staging for a client with a 32 MB client file cache. Each set of bars shows the cumulative delay due to file system activity for a different client trace. The first two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches.

Figure 6. Benefits of data staging with 32 MB client cache

5.3.2 Results

For these traces, we used the T20 as the file system client; we are currently duplicating the experiments with the iPAQ handheld as client. We first replayed the traces assuming a client file cache size of 32 MB—Figure 6 shows the results. Data staging provides significant benefits for all traces under both scenarios. Under the `warm` scenario, data staging reduces the time to perform file operations by an average of 62%. The maximum reduction of 71% occurs for the `robin` trace—the minimum of 47% for the `berlioz` trace. Reductions under the `cold` scenario are slightly less, averaging 59%.

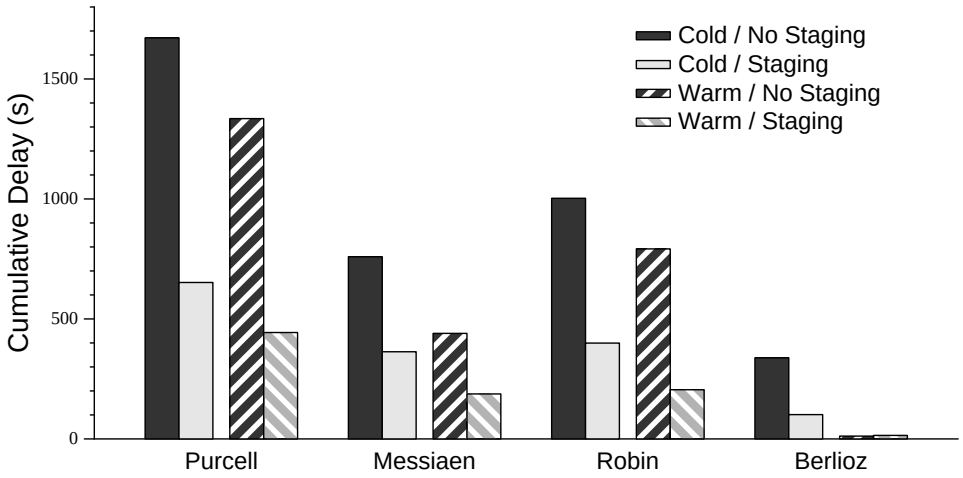
Figure 7 shows results when we increase the client file cache to 64 MB. The most noticeable difference occurs for the `berlioz` trace under the `warm` scenario. The entire working set of the trace fits in the client file cache—there is no need to fetch data from the server since we assume perfect prediction. However, as the `cold` scenario demonstrates, if one is unable to perfectly predict what files will be fetched, some high-latency fetches will be needed even for a small working set, and data staging can be of benefit. The average reduction in cumulative delay for file operations with a 64 MB cache is 61% in the `cold` scenario and 54% in the `warm` scenario.

Figure 8 shows more detailed results for a representative trace—`purcell` with a 64 MB client file cache. Each line represents the cumulative fraction of file operations that complete within a given time period. For this trace, over 80% of the file operations are reads that hit in the client file cache or writes that are buffered on the client for later reintegration—these incur negligible delay.

The shape of the `Cold / No Staging` line confirms one of our assumptions: latency, not bandwidth, is the real killer in distributed file system performance. Of the operations which do not complete immediately, the vast majority of operations take very close to the 60 ms. round-trip delay. There is also a noticeable cluster of operations around 120 ms.—this is caused by operations such as directory reads for which Coda requires two round-trips. Data staging significantly shortens the vast majority of these high latency operations. For interactive applications that incur the cost of several sequential file operations, this dramatically reduces the frustrating delays experienced by the user.

6 Related Work

This work is one of the first to focus on how untrusted hardware can be used to improve distributed file system performance for small, storage-limited clients without compromising security. Yet, it builds upon a large body of previous work in caching and speculative prefetching.



This figure shows the benefits of data staging for a client with a 64 MB client file cache. Each set of bars shows the cumulative delay due to file system activity for a different client trace. The first two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches.

Figure 7. Benefits of data staging with 64 MB client cache

At a conceptual level, data staging shares several goals with edge computing initiatives such as distributed Web caching. Companies such as Akamai [1] have developed content distribution networks which push data towards end nodes to reduce access latency. However, data staging's focus on file data creates many important differences. Consistency is a first-class concern for us: data staging preserves the consistency guarantees of the underlying distributed file system. Also, file data is far less likely to be shared among multiple users than Web data. Finally, privacy and authenticity are first-class concerns; we provide a mechanism for end-to-end encryption that avoids the need to trust third parties. These issues also differentiate surrogates from caching Web proxies such as Squid [18].

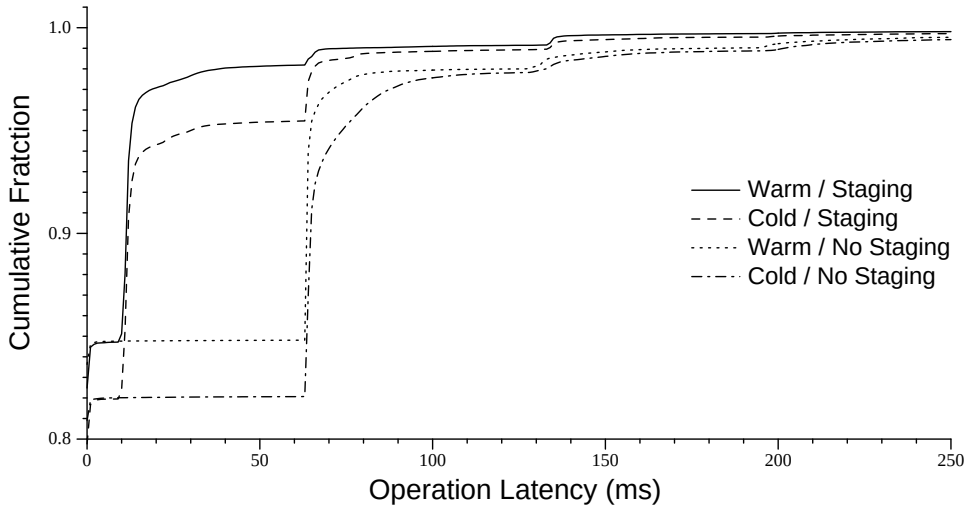
WayStations in Fluid Replication [7] perform a role similar to that played by our surrogates. Yet, there are several key differences. First, replicas stored on WayStations accept file modifications from clients. In data staging, modifications are transmitted directly to the server, simplifying the trust model for our surrogates. A second difference is that WayStations do not speculatively prefetch data, and thus may perform poorly in cold-cache scenarios.

OceanStore [10] provides floating replicas of data which can migrate to nearby servers. OceanStore partitions servers into those trusted to perform replication protocols and those not trusted to do so. While the untrusted servers may optimistically accept updates, the client must directly contact the trusted servers in order to ensure permanence and correctness.

Several algorithms for predicting what data to cache have been proposed, for example SEER [11] in the file system arena, and distributed cache placement algorithms [20] in the Web arena. This body of work has a nice synergy with data staging: in the future, we hope to explore how predictive strategies other than LRU affect hit rates at surrogates.

Infostations [23] have been proposed as a mechanism for providing high-bandwidth access to data in mobile environments. Data is aggressively transmitted to clients as they pass through short-range, high-bandwidth zones. Although our work has focused on high-latency connections, we believe that the data staging architecture would be an ideal mechanism for caching file data at Infostations in preparation for burst transmission to clients.

Muntz and Honeyman [13] evaluated the use of intermediate caching for the AFS file system and found that it achieved little benefit. However, their evaluation environment is quite different from the target environment for data staging. Wide-area network latency imposes substantially greater penalties for cache misses. In addition, our clients are assumed to be storage-limited, and we do not focus on sharing files among multiple users.



This figure shows how data staging reduces file operation latency for the `purcell` trace with a 64 MB file cache. Each line shows the cumulative fraction of file accesses that finish on or before the indicated time.

Figure 8. Reduction of file operation latency for `purcell` trace

In effect, data staging applies the well-understood concept of prefetching [3, 15] to pervasive computing environments. Instead of prefetching file blocks from the disk, data staging prefetches whole files from a distant server.

7 Conclusion

Untrusted and unmanaged machines can facilitate mobile data access. Data staging uses nearby surrogates located in the pervasive computing environment to improve distributed file system performance for storage-limited clients. Clients borrow storage capacity from surrogates and use it as a second-level file cache to hide the latency of file operations.

Two important assumptions in our work are that surrogates are untrusted and unmanaged. Because surrogates are untrusted, we use end-to-end encryption to provide privacy, and secure hashes to ensure authenticity. We make surrogates as reliable and easy to manage as possible by maintaining no hard state on them, using commodity software, and pushing functionality from surrogates to client and server machines. We believe these design considerations will prove vital in ensuring the wide-spread deployment of a surrogate infrastructure.

References

[1] AKAMAI CORPORATION. *http://www.akamai.com*.

[2] CARSON, M. *Adaptation and Protocol Testing thorugh Network Emulation*. Internetworking Technologies Group, NIST, http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm, September 1999.

[3] CHANG, F., AND GIBSON, G. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 1–14.

[4] DOUCEUR, J., AND BOLOSKY, W. A Large-Scale Study of File-System Contents. In *Proccedings of the the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Atlanta, GA, May 1999), pp. 59–70.

[5] FARKAS, K. I., FLINN, J., BACK, G., GRUNWALD, D., AND ANDERSON, J. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)* (Santa Clara, CA, June 2000).

[6] GARLAN, D., SIEWIOREK, D., SMAILAGIC, A., STEENKISTE, P. Project Aura: Toward Distraction-free Pervasive Computing. *IEEE Pervasive Computing 1*, 2 (April-June 2002).

[7] KIM, M., COX, L. P., AND NOBLE, B. D. Safety, visibility, and performance in a wide-area file system. To appear in Proceedings of the 1st USENIX Conference on File and Storage Technologies, January 2002.

[8] KISTLER, J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[9] KROEGER, T., AND LONG, D. The case for efficient file access pattern modeling. In *Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)* (Rio Rico, AZ, March 1999).

[10] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, November 2000), pp. 190–201.

[11] KUENNING, G., AND POPEK, G. Automated hoarding for mobile computers. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint-Malo, France, October 1997).

[12] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995).

[13] MUNTZ, D., AND HONEYMAN, P. Multi-level caching in distributed file systems or Your cache ain't nuthin' but trash. In *Proceedings of the Winter 1992 USENIX* (January 1992), pp. 305–313.

[14] NARAYANASWAMI, C., KAMIJOH, N., RAGHUNATH, M., INOUE, T., CIPOLLA, T., SANFORD, J., SCHLIG, E., VENKITESWARAN, S., GUNIGUNTALA, D., KULKARNI, V., YAMAZAKI, K. IBM's Linux Watch: The Challenge of Miniaturization. *IEEE Computer 35*, 1 (January 2002).

[15] PATTERSON, R., GIBSON, G., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995).

[16] SATYANARAYANAN, M. Pervasive computing: Vision and challenges. *IEEE Personal Communications 8*, 4 (August 2001).

[17] SATYANARAYANAN, M. The Evolution of Coda. *ACM Transactions on Computer Systems 20*, 2 (May 2002).

[18] *Squid Web Proxy Cache*. http://www.squid-cache.org/.

[19] STEMM, M., AND KATZ, R. H. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Science, Special Issue on Mobile Computing 80*, 8 (August 1997), 1125–1131.

[20] VENKATARAMANI, A., WEIDMANN, P., AND DAHLIN., M. Bandwidth constrained placement in a wan. In *20th ACM Symposium on Principles of Distributed Computing (PODC)* (August 2001).

[21] VOGELS, W. File system usage in Windows NT 4.0. In *Proccedings of the 15th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, December 1999), pp. 93–109.

[22] WALDO, J. The Jini architecture for network-centric computing. *Communications of the ACM 42*, 7 (1999).

[23] WU, G., CHU, C.-W., WINE, K., EVANS, J., AND FRENKIEL, R. WINMAC: A Novel Transmission Protocol for Infostations. In *Proceedings of IEEE VTC* (Houston, TX, 1999).